DOCUMENT RESUME

ED 230 169 IR 010 681

AUTHOR	Eisenstadt, Marc; And Others
TITLE	Creating Pleasant Programming Environments for Cognitive Science Students. Technical Report No. 16. [and] Domain Specific Debugging Aids for Novice Programmers. Technical Report No. 17.
INSTITUTION	Open Univ., Walton, Bletchley, Bucks (England).
PUB DATE	Aug 81
NOTE	11p.; Produced by the Computer Assisted Learning Research Group.
PUB TYPE	Reports - Descriptive (141) -- Speeches/Conference Papers (150)
EDRS PRICE	MF01/PC01 Plus Postage.
DESCRIPTORS	*Computer Programs; *Computer Science Education; *Databases; Design Requirements; Instructional Materials; Models; Postsecondary Education; *Programing; *Programing Languages
IDENTIFIERS	Debugging (Computers); *Debugging Aids; Knowledge; Open University (Great Britain); SOLO Programing Language; *User Cordial Interface

ABSTRACT

These papers describe efforts toward the systematic improvement of a software environment called SOLO (Eisenstadt, 1978), which has been used with cognitive psychology students at the Open University in Great Britain and is geared towards the manipulation of assertional databases. The first paper, by Eisenstadt, Laubsch, and Kahney, provides an introduction and background information about SOLO, followed by descriptions of empirical studies undertaken to analyze students' programming behavior, which have highlighted the use of a small number of programming schemas by a large proportion of students. These schemas serve as a basis for an automated debugging assistant, which is also described. Seven references are listed. The second paper, by Laubsch and Eisenstadt, describes recent progress in the design and implementation of the debugging assistant which was designed for novice programmers learning to use a simple assertional database language. Oriented towards a large audience of computer-naive users, this system deals with argument passing, recursion, and side-effects. Specific topics discussed are (1) deriving effect descriptions by symbolic evaluation; (2) matching the effect description with a domain-specific library plan; and (3) comparing effect descriptions of user plans with library plans. Ten references are listed. (Author/LMM)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED230169

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☐ This document has been reproduced as received from the person or organization originating it.

☑ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official NIE position or policy.

CREATING PLEASANT PROGRAMMING ENVIRONMENTS
FOR COGNITIVE SCIENCE STUDENTS
Technical Report No. 16
Marc Eisenstadt, Joachim Laubsch, & J. H. Kahney

DOMAIN SPECIFIC DEBUGGING AIDS
FOR NOVICE PROGRAMMERS
Technical Report No. 17
Joachim Laubsch and Marc Eisenstadt

The Open University
Computer Assisted Learning Research Group

IR010681

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY

Marc Eisenstadt

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

CREATING PLEASANT PROGRAMMING ENVIRONMENTS FOR COGNITIVE SCIENCE STUDENTS

M. Eisenstadt, J.H. Laubsch, & J.H. Kahney

The Open University
Milton Keynes, ENGLAND*

1. Introduction and background

This paper describes our current efforts towards the systematic improvement of a LOGO-like software environment called SOLO (Eisenstadt, 1978), which has been used by over 1500 Cognitive Psychology students at the Open University. SOLO is geared towards the manipulation of assertional data bases, and provides the students with a handful of easy to use primitives with which they can address some elementary problems of knowledge representation. Students login to one of our DECsystem-20s from a regional study centre, and experience about 10 hours of hands-on activity early in the academic year. Later on, they attend a residential summer school at which they can acquire an additional 30 hours of hands-on experience.

Although SOLO is a toy language in some respects, the total user environment has many features which make it ideal for providing the vast majority of our students (80% of whom are computer-naive) with their first exposure to computing. Among these features are a spelling corrector, syntax-directed editing aids, automataic display of data base changes as they occur, "undo" facilities, and an easily modifiable user-profile.

An in-depth analysis of our students' errors (Lewis, 1980) has led to an improved design to help ensure that beginners can write syntactically correct programs with a minimum of fuss. A microcomputer implementation, which uses screen-oriented syntax-directed editing (cf. Teitelbaum & Reps, 1980) is being piloted just prior to this conference.

Even with SOLO's extensive user aids and carefully pre-tested curriculum materials, our students still experience problems in writing programs which perform precisely as intended. Because of this, we have undertaken a detailed analysis of their programming behaviour. Our empirical studies, described in section 2, have highlighted the use of a small number of programming schemas by a large proportion of our students. These schemas serve as the basis for an automated debugging assistant, which is described in section 3.

---

* This research is supported by grants from the U.K. Social Science Research Council (HR7372/1) and the German Academic Exchange Service. This paper appears in the Proceedings of Third Annual Cognitive Science Society Conference, Berkeley, CA, U.S.A., August 1981.

2. The behaviour of SOLO programmers

As part of their SOLO activities at the beginning of the year, our students are asked to "write a program which makes the following inference: If someone is found to be guilty, then whoever that person works for is also guilty." In solving the problem, students are invited to invent their own data structures and algorithms. We analyzed a sample of 160 student programs to see if some underlying order could be found among a potentially large variety of databases and program structures. As it turns out, the programs written by these students are built from a handful of basic program schemas. These schemas are langauge-independent programming constructs such as FILTER, CONJUNCTION, SIDE-EFFECT, and GENERATE-NEXT-OBJECT, which are closely related to those found in the LISP "plan library" of Shrobe, Waters & Sussman (1979).

The students' databases can be broken down into basic relational patterns. These patterns are reliably associated with particular program structures, allowing us to predict in 80% of the cases precisely what the students' program organization will be. To illustrate this point, consider the following typical (student-generated) database:

```
BURGESS                      PHILBY
 |---ISA--->COMMUNIST         |---ISA--->COMMUNIST
 |---WASAT--->CAMBRIDGE       |---WASAT--->CAMBRIDGE
 |---WORKSFOR--->PHILBY       |---WORKSFOR--->BLUNT

BLUNT
 |---ISA--->COMMUNIST
 |---WASAT--->CAMBRIDGE
 |---WORKSFOR--->THEQUEEN
```

This database exhibits the following patterns:

```
Transitive-Relation:
      WORKSFOR (BURGESS PHILBY BLUNT THEQUEEN)

Shared-Successor:
      (ISA COMMUNIST)    (BURGESS PHILBY BLUNT)
      (WASAT CAMBRIDGE)  (BURGESS PHILBY BLUNT)
```

Several items (BURGESS, PHILBY, BLUNT) are present in both the Transitive-Relation and the Shared-Successor lists. One of the program structures typically accompanying such a database structure contains three segments: a CONJUNCTION (COMPLEX-FILTER), SIDE-EFFECT, and GENERATE-NEXT-OBJECT. The Shared-Successors will be used as a FILTER selectively to choose nodes on which a SIDE-EFFECT is perpetrated (e.g. asserting X ISA SPY) and the Transitive-Relation list will be used to GENERATE-NEXT-OBJECT. Since the Shared-Successor list

BEST COPY AVAILABLE

contains two patterns, a COMPLEX-FILTER will almost surely be used: IF x isa communist AND x wasat cambridge THEN ...

But why should a student write a program like this in the first place? Our analysis indicates that students have their own stylised interpretations, or mental models, of the task at hand. For instance, some students think that a program involving two inferences is called for: "If X has done something criminal then he is guilty. And if this is so, then his employer is subject to the same scrutiny, and so on for all employers." Other students feel that only one inference is called for: "Assuming X is guilty, his employer, by association, is guilty also, and so on for all employers."

The observed program structures ought to correspond to students' mental models of the task. Some of these mental models are "appropriate", in that they address the problem as stated, while others introduce certain anomalies which preclude a satisfactory solution. Such "inappropriate" models could actually be artifacts of students "thinking in SOLO" and getting led astray.

In order to test these ideas we have begun studying individual students in depth, collecting videotaped protocols and terminal session transcripts. The first subject began her project session with the clear intention of writing a program involving two inferences. Because of preconceived and inaccurate notions about constraints on the way she was allowed to approach the problem and because of misconceptions arising from her interaction with SOLO, she twice altered her intentions. At the end of the session the student had a working program for a "one-inference" interpretation of the task described above, a compromise with which she herself was not completely satisfied. All of her programming behavior throughout this session, her various approaches to solving the problem, and the bugs she encountered, fell within the scope of the structures we had identified in the earlier analysis of 160 programs.

Our ability to categorize standard database structures and predict implementation strategies on the basis of those structures means that we can develop tools for assisting students in terms of the way in which they prefer to think about the task at hand. One such tool is described in the next section.

3. Debugging Aids

We have designed and partially implemented a tutorial debugging assistant which attempts to articulate the causes of bugs in terms which are close to the way we believe the students actually think about their own implementations. The bugs dealt with range from domain independent violations of the semantics of SOLO to domain specific errors that can be detected only if knowledge about the task at hand is used.

The debugging assistant uses symbolic evaluation (cf. Ruth, 1976) as a tool for (1) recognizing procedures as parts of a given "library plan", (2) detecting errors of the following types: unreachable program steps, purposeless steps, reference to absent database objects, infinite recursion because of a missing or unsatisfiable termination condition.

In the tutorial situation, a student's goal is to write a program to accomplish some modelling task. The debugging assistant is provided with a prototypical solution in terms of a canonical effect description. The task of the assistant is to recognize a match between the canonical effect description and an effect description derived from the student's own program.

In general these will not match, and the nature of the deviation will enable the assistant to draw the student's attention to shortcomings of his or her program which may be classified in the following way:

- The program will achieve the desired effect only in certain cases. A counter-example outside this range can provoke the student to discover the cause.

- The program would work if missing data or inconsistent entries in the data base were corrected. These corrections can be pointed out directly.

- A particular sub-procedure, if corrected using heuristics about typical errors (e.g. missing indirect link, violation of a program schema), would make the overall program correct. In this case, an appropriate hint can be provided for the student.

- None of the above.

In the last case, the student may initiate a dialogue, requesting help on a particular procedure. During the dialogue the assistant tries to find out the intended effect of that procedure (Eisenstadt & Laubsch, 1980). It does this by

BEST COPY AVAILABLE

categorizing the procedure into one of several programming schemas stored in a language-independent "plan library", using a notation developed by Rich & Shrobe (1978).

Consider the case in which the nearest matching schema is "conjunctive filter and side effect". The assistant examines the deviation between the user's procedure and the stored schema. The following violations of the use of that schema may be recognized: omission of a conjunct, omission of the side effect, wrong (or transposed) arguments in the slots of the schema, or wrong control flow links. The assistant can describe these violations in terms which the student can relate to, since the library plans are themselves based upon schemas known to occur in students' code.

Since the students' procedures depend on their databases, and vice versa, the debugging assistant relies heavily on domain-specific knowledge to deal with alternative ways of formulating a solution to a given problem. Although the students have a great deal of freedom to choose ways of implementing solutions, they typically resort to a few common approaches. The assistant knows about these, and uses these both to make sense of what the students are attempting and to explain why they have gone astray.

4. Conclusions

Our experience with SOLO leads us to believe that a SOLO-like language/environment/curriculum could be of use to a broader group of cognitive science students-- for instance as the basis of a beginners' LISP curriculum oriented entirely towards pattern-matching and assertional data bases. For this to become a reality, it is important to understand precisely what remaining problems students have in this type of environment, and why. Our empirical work is a step in this direction. It has immediate spinoffs in that it provides a foundation for our debugging assistant. The assistant provides students with a tool for attaining their goals, and provides us with a tool for analyzing and describing their behaviour.

REFERENCES

Eisenstadt, M. Artificial intelligence project. Units 3/4 of Cognitive psychology: a third level course. Milton Keynes: Open University Press, 1978.

Eisenstadt, M. & Laubsch, J. Towards an automated debugging assistant for novice programmers. Proceedings of the AISB-80 conference on Artificial Intelligence, Amsterdam, 1980.

Lewis, M. Improving SOLO's user-interface: an empirical study of user behaviour and proposals for cost-effective enhancements to SOLO. Technical report no. 7, Computer Assisted Learning Research Group, The Open University, 1980.

Rich, C. & Shrobe, H. Initial report on a LISP programmer's apprentice. IEEE Transactions on Software Engineering, SE-4:6, 1978.

Ruth, G.R. Intelligent program analysis. Artificial intelligence, 7, 1976.

Shrobe, H., Waters, R., & Sussman, G. A hypothetical monologue illustrating the knowledge underlying program analysis. MIT Artificial Intelligence Laboratory Memo 507, 1979.

Teitelbaum, T., & Reps, T. The Cornell program synthesizer: a syntax-directed programming environment. Technical report 80-421, Department of Computer Science, Cornell University, 1980.

BEST COPY AVAILABLE

DOMAIN SPECIFIC DEBUGGING AIDS FOR NOVICE PROGRAMMERS

Joachim Laubsch and Marc Eisenstadt

The Open University
Milton Keynes, ENGLAND*

ABSTRACT

We have been developing a debugging aid tailored to novice programmers learning to use a simple assertional database language. Based in part on existing programmer's apprentice and debugging projects, our system makes several novel contributions: it is oriented towards a large audience (several hundred per year) of computer-naive users; the system deals with argument passing, recursion, and side-effects. Symbolic evaluation of the user's code produces an effect-description which is compared with an idealised effect-description (derived from a library plan). The library plan uses domain-specific knowledge to allow the user a great deal of freedom to invent alternate approaches to the programming task at hand. Based on effect-description mismatches, a variety of hints for correcting the code can be given to the student.

I INTRODUCTION

This paper describes recent progress in the design and implementation of a debugging assistant for novice programmers. The novices are Open University psychology students learning about problems of modelling and representation using SOLO, a LOGO-like language oriented towards the manipulation of an assertional data base [2]. To give the reader a quick feel for SOLO, here is a short SOLO program and its MacLisp counterpart:

```
SOLO:   TO INFECT /X/
        1 NOTE /X/ HAS FLU
        2 CHECK /X/ KISSES ?Y
          2A If present: INFECT *Y; EXIT
          2B If  absent: PRINT "ADIOS"; EXIT

Lisp:   (defun infect (x &aux y)
          (assert `(,x has flu))
          (cond ((present `(,x kisses ?y))
                 (infect y))
                (t  (print "ADIOS"))
          nil)
```

Fig. 1

Extensive user aids ensure that students enter lexically and syntactically correct programs. Even so, the programs typically fail to perform as intended. While realising that such failures can be of great pedagogical value, our aim is to develop a technique for helping users circumvent hurdles which they might find too tedious to deal with or which may prevent them from attacking pedagogically more valuable tasks. Since our students normally work on their own (via dial-up lines to one of our DECsystem-20's), the provision of intelligent debugging aids is of great practical benefit.

Students write programs on one of several suggested topics. Schema-matching, simple inferencing, and children's subtraction skills are among the favourites. The subtraction task is particularly interesting, because SOLO has no numerical primitives, and thus students must think very carefully about how to represent numbers in a database, what operations to allow, etc. We can provide our debugging assistant with a fair amount of knowledge about subtraction, which makes it a good testbed for our ideas. It is important to note, however, that we do not expect the students to have to work towards implementing a particular algorithm. On the contrary, one of the novel aspects of our approach is that we encourage the students to come up with their own representations and algorithms. The data representation they choose will of course influence the algorithm they use, and vice versa. Our debugging assistant is provided with detailed knowledge of a wide range of possible data structures which could be used to represent subtraction tables, along with an abstract description of effects to be achieved by a subtraction program. Only after examining the student's representation does the assistant decide what the 'idealised effect description' for that student's code should be-- i.e. what changes to the assertional data base ought to occur as a result of a successful run of his program.

The debugging assistant deals with parameterised subroutines, conditionals, and side effects, thus extending the pioneering work of Adam and Laurent [1], Goldstein [5], Miller and Goldstein [6], and Ruth [8]. It relies extensively on a language-independent 'plan diagram' representation developed by Rich and Shrobe [7], and Waters [10]. The latter is the basis for computing the effects of programs and program segments, which in turn gives the debugging assistant a way to understand what the program

* This research is supported by grants from the U.K. Social Science Research Council (HR7372/1) and the German Academic Exchange Service. This paper appears in the Proceedings of the Seventh International Joint Conference on Artificial Intelligence, Vancouver, BC, Canada, August 1981.

BEST COPY AVAILABLE

achieves, and a way to compare this with what the program should have achieved. In the case of small mismatches, it is possible to recommend a particular patch to the student. More importantly, however (since we don't want to write the entire program for the student) is the generation of tailor-made counterexamples to provoke the student into thinking up his or her own solution.

When a student gets stuck, he or she can invoke the debugging assistant by typing HELP DEBUG. The assistant investigates the student's code and data base in several phases, as outlined below:

* TRANSLATION: User programs are translated (as they are entered) into a language independent 'plan diagram' notation (PDN) [7, 9, 10] which encodes the control flow, the data flow, and a declarative description of each program's overall effect. During translation, three kinds of 'irrational code' can be detected (cf. [5]): (1) unreached code; (2) an unbound CHECK variable used following the 'absent' branch of a CHECK statement; (3) useless code, which includes variable binding without later reference, assertion and deletion of the same data base triple, and superfluous CHECKs. The translation phase is described in [3].

* PLAN RECOGNITION: Certain sequences of code (some of which are specific to SOLO, others of which are specific to the subtraction task) can be recognized in their PDN form as manifestations of various 'standard library plans'. This recognition is performed by referring to a plan library, which knows not only how to recognize plans, but also what their effect descriptions are. We have identified over a dozen standard library plans which are manifest in the code of actual SOLO users. A typical plan is 'simple net traversal'. In terms of the plan library of Shrobe et. al. [9], this is a 'generator' plan, which recursively fetches all the nodes along a chain of a particular relation, e.g. nodes n1, n2 and n3 in the network below:

n1---REL--->n2---REL--->n3

This in turn can be used as part of a more complicated plan, such as 'generate and side-effect each'. This plan is the basis for the INFECT program shown in Fig. 1-- INFECT generates a sequence of nodes by simple net traversal (steps 2 and 2A), and unconditionally side-effects every one of them (step 1). The role of such plans is discussed in [4].

* SYMBOLIC EVALUATION: An analysis is made of all possible pathways which the program might have explored, and an effect description is generated, providing a declarative account of what the program achieves in terms of changes to the data base. This is especially important for those pieces of the program which did not appear to match any known plans in the plan library. This phase is described in section II below.

* EFFECT DESCRIPTION MATCHING: The effect description of the student's program is now compared with an idealised effect description to find mismatches. The idealised effect description is computed by expanding a domain specific library plan in a top-down fashion to mesh with the student's own chosen data base representation (which we analyse bottom-up). The debugging assistant has criteria for detecting near misses, and heuristics for suggesting how to repair a program. If this fails, the effect descriptions can still provide the basis for generating useful counterexamples which may help the student to focus more clearly on the underlying cause of the bug. Section III discusses effect description matching in some detail, and gives examples of the kinds of bugs which our system can currently deal with.

All of the examples shown below run in our MacLisp implementation, although the debugging assistant is not yet robust enough for use in a 'live' student environment.

II DERIVING EFFECT DESCRIPTIONS BY SYMBOLIC EVALUATION

A. Symbolic evaluation describes a program's effect by stepping through all cases

Symbolic evaluation serves as a tool for deriving a canonical description of the effect that a user program will have, when run in some environment. This description can be compared with the effect description of known (library) plans. Such a comparison will show that the program either behaves as intended by the tutor, deviates "slightly" (in which case we can suggest a patch), or deviates "grossly" (in which case we can derive counter-examples).

SOLO programs typically modify a global data base by side-effects. A procedure receives arguments and implicitly an input data base (db-in), and produces an output data base (db-out). The effect of a program is described by a tree which at its root has db-in and at its leaves has db-out for each possible condition.

The data base consists of a set of triples, each having the form

triple = [<source> <link> <target>]

We represent a data base as an association list of bindings, where each binding is

binding = [bind (<source> <link>) <target>]

The original data base, after the student has used a problem-specific SETUP procedure (and possibly after a series of top-level assertions) can then be represented as

db0 = {[bind (<source1> <link1>) <target1>]...}

By "the binding of (a b) in e", abbreviated "(get a b)", we refer to the target of the first binding whose left part is (a b) in a data base e. Alternative environments are represented as a cases

BEST COPY AVAILABLE

statement of the form:

(cases [<predicate1> <environment1>] ...)

The predicate can refer to node identity, e.g. (= (get a b) c)), or to the presence of a binding, e.g. (present <source> <link>). We interpret the latter predicate as true in an environment e if (<source> <link>) is bound in e.

B. Effect descriptions for SOLO primitives

Each primitive step in a plan diagram is an instance of some class which has associated with itself the knowledge of how it affects the data base. Symbolic evaluation produces the following: (1) augmentations to db-in (for FORGET and NOTE steps); (2) a local binding environment; (3) a cases statement (for CHECK steps).

CHECK steps come in two varieties: with an open or a closed variable as the target. A CHECK with an open variable produces a local binding to that variable. The local binding environment is not shown in the effect description since it does not change the data base. Instead, the binding environment is used for finding the substitution of a variable in terms of access to the data base through the procedure's input parameters.

Here is an example taken from a definition used in a student's subtraction program. It looks up the difference between two numbers /A/ and /B/ by consulting some table T (pointed to by the relation 'SUBTBL' emanating from node /A/). The result, R, is stored on the node /ANS/. The student could have chosen to represent subtraction tables in the form of simple relations, e.g. 5---4--->1, 5---3--->2, etc., but has instead opted for a more flexible representation of the form 5---SUBTBL--->5SUB, 5SUB---3--->2. This allows addition and multiplication tables to be stored in the data base as well. The use of indirect accessing of tables gives the procedure its name, 'Diff-ind':

```
TO DIFF-IND /A/ /B/ /ANS/
1 CHECK /A/ SUBTBL ?T
  1A If present: CONTINUE
  1B If absent:  EXIT
2 CHECK *T /B/ ?R
  2A If present: CONTINUE
  2B If absent:  EXIT
3 NOTE /ANS/ VAL *R
```

Fig. 2

For the purposes of symbolic evaluation, here are the key events which occur in the above procedure:

* At step 1A, T is bound to (get /A/ SUBTBL).

* At step 2A, R is bound to (get (get /A/ SUBTBL) B).

* At step 3, db-in is augmented by [bind (/ANS/ VAL) (get (get /A/ SUBTBL) /B/)].

A CHECK-step produces two possible environments, which are represented as alternative branches of a cases statement. This yields the following effect description for DIFF-IND in environment e:

```
1 diff-ind(/a/ /b/ /ans/) =
2  (cases [(present /a/ SUBTBL)
3          (cases [(present (get /a/ SUBTBL) /b/)
4                  {[bind (/ans/ VAL)
5                         (get (get /a/ SUBTBL)
6                              /b/)] . e } ]
7                  [t e])
8          [t e])
```

Fig. 3

It is possible to avoid the split into alternative cases if the (<source> <link>) pair of the CHECK is known to be present or absent in the current environment. This occurs if a case predicate further up in the tree (possibly at db0) implies this. For example, in the case of an invocation of DIFF-IND in an environment where both case predicates are known to be true (i.e. /A/'s SUBTBL link and that table's /B/ link are both known to be present), the above effect description would be simplified to:

```
[[bind (/ans/ VAL)
       (get (get /a/ SUBTBL) /b/)] . e ]
```

A CHECK with a closed variable or constant as target is translated as follows:

```
(CHECK (a b C) <present-code> <absent-code>)  =>

      (cases [(= (get a b) C) <present-code>']
             [t <absent-code>']])
```

where <form>' is the translation of <form>.

A step containing a call to a user-defined procedure in some environment db-in is translated as:

```
(<user-proc-name> <arglist>) =>
    (db-out (<user-proc-name> <arglist> db-in))
```

Here (db-out <form>) denotes the output environment produced by the symbolic evaluation of <form> which will be expanded in-line.

A useful rule for simplification of user procedure calls is:

```
(db-out (<user-proc-name>
         <arglist>
         [<binding> . <env>]))    =>

[<binding>
 (db-out (<user-proc-name>
          <arglist>
          <env>))]
```

which is applied whenever it can be shown that the user procedure does not depend on <binding>.

BEST COPY AVAILABLE

III MATCHING THE EFFECT DESCRIPTION WITH A DOMAIN-SPECIFIC LIBRARY PLAN (DSLP)

The typical output of symbolic evaluation will be a highly nested cases statement with get-compositions embedded in the predicates and environment. Unless we can assign "meaning" to these get-compositions, we cannot interpret this as a solution to the student's task, since the student may have chosen from a variety of data representations, each requiring special access and data base change operations. To take a rather simple example, the student may have asserted triples like [9 COMPL 1], [8 COMPL 2], etc. If we knew that these triples represented a 10-complement function, we could parse (get x COMPL) as (10-complement x) for any x. Our assistant has a repertoire of low level DSLPs which extract this knowledge from the student's data base for each possibly useful function or predicate in the subtraction domain (e.g. addl, subl, val-of, geq).

Once these low level DSLPs are recognized, we can do a top-down analysis similar to that of Ruth [8], starting from a high level DSLP which is predefined by us for each particular task. We will now focus on a typical task undertaken by our students, namely writing a SOLO program to perform two column subtraction. The student is given definitions of SETUP and ANSWER procedures which impose a layout of the form:

```
A        B       |
                 |        SETUP
C        D       |
-----------------
E        F       |        ANSWER
```

The DSLP for this task is like a grammar that can be used to recognize all possible variations in the actual effect description of the user's code. For 2 column subtraction the DSLP has the following definition (omitting db-out for brevity):

```
1 2-COL-SUB (A B C D E F db-in) =
2  (cases [(geq (val-of B) (val-of D))
3          (cases
4           [(geq (val-of A) (val-of C))
5            {[bind (val-of E)
6                   (diff (val-of A) (val-of C))]
7             [bind (val-of F)
8              (diff (val-of B) (val-of D))]
9             . db-in}]
10          [t (fail 2-COL-SUB)])]
11         [t {[bind (val-of E)
12              (borrowed-from
13                (val-of A) (val-of C))]
14             [bind (val-of F)
15              (borrowed-by (val-of B) (val-of D))]
16             . db-in}])
```

Fig. 4

The student may have written the following program for two column subtraction:

```
TO SUBTRACT
1  DIFF-IND B D F
2  CHECK BORROW-FLAG IS SET
   2A If present: DECREMENT A; CONTINUE
   2B If absent: CONTINUE
3  DIFF-IND A C E
4  ANSWER
```

Fig. 5

The assistant recognizes this as a function at the top of the definition hierarchy and symbolic evaluation expands it by inserting the effect descriptions of the called functions (and so on recursively for these). The resulting effect description is a large composition of gets, which ideally should be matched by 2-COL-SUB.

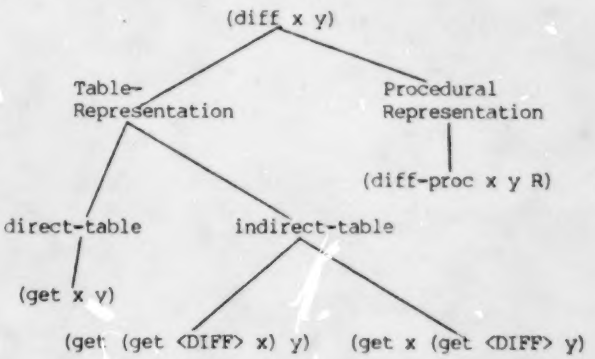
Recognition now proceeds top-down, involving the sub-plans val-of, geq, diff, borrowed-from and borrowed-by. Each of these sub-plans is itself rewritten to conform with the particular representation adopted by the student. E.g. the sub-plan for dealing with a borrowed-from column containing the digits x and y can be re-written in two ways:

```
borrowed-from (x y) = (diff (subl x) y) |
                      (diff x (addl y))
```

Similarly, the diff plan (which implements x - y for x ⩾ y) must be rewritten in terms of primitives which the student may have elected to use. The difficulty here is that the student may have chosen to implement 'diff' in terms of tables in the data base or a recursive counting procedure. We will consider both representations:

1. The student may have invented a 2-digit subtraction table

He may have chosen among a wide variety of table representations. We represent these as a class hierarchy, which for each class has the associated rule for rewriting the user's effect description into diff. <DIFF> is an arbitrary symbol invented by the user.



If we take account of argument reversals, there are 10 possible ways of organizing subtraction tables. Each indirect-table subclass has two triples as parts which are mutually constrained (the outer and the inner get) by the difference predicate. We hypothesize a particular table-class,

BEST COPY AVAILABLE

and seek confirmation until we can decide on the actual class used by the student. Then we look for all the entries of the table and create an object DIFF which has associated with it the code to recognize (diff a b), e.g. as (get (get a SUB) b). Data base analysis reveals the frequent error-type of missing or inconsistent entries in the data base. Triples which belong to no pre-defined class, but "partially" match some class (i.e. because they contain a known link-name) are categorized as inconsistent and the user is informed about them.

2. The student may have represented 'diff' procedurally

Using addl and subl, he may have written a recursive procedure to implement 'diff'. Since a SOLO procedure can only have an effect by side-effecting the data base, the DSLP for a procedural implementation has to change slightly. Any function f, which is represented procedurally as f-proc will be transformed in the library plan as follows:

```
{[bind (val-of x) (f <arglist>)] . e} =>
      (f-proc <arglist> N e))
```

where N is the node to be side-effected.

The following example shows a typical student implementation of 'diff' together with the effect description produced by the debugging assistant:

```
TO DIFF-REC /X/ /Y/ /ANS/
1 CHECK EQUALS /X/ /Y/
 1A If Present: NOTE /ANS/ VAL 0; EXIT
 1B If  absent: CONTINUE
2 CHECK /X/ SUB1 ?x1
 2A If present: DIFF-REC *x1 /Y/ /ANS/; CONTINUE
 2B If  absent: EXIT
3 CHECK /ANS/ VAL ?v
 3A If present: CONTINUE
 3B If  absent: ERROR; EXIT
4 CHECK  *v ADD1 ?v1
 4A If present: NOTE /ANS/ VAL *v1
 4B If  absent: ERROR; EXIT
```

Symbolic evaluation first produces:

```
diff-rec (x y ans db-in) =
 (cases [(= (get EQUALS x) y)
         {[bind (ans VAL) 0] . db-in}]
        [(present (get x SUB1))
         {[bind (ans VAL)
                (get (get ans VAL) ADD1))] .
          (self (get x SUB1) y ans db-in)}])
```

This matches a recursion pattern for which there is a DSLP. The binding being affected in the terminating case (ans VAL) is identical to that of the post-recursive step, which means that (ans VAL) is successively rebound. The DSLP simplifies this to an instance of a prototypical data base entry:

```
[db-entry value   [bind (ans VAL) g007]
          init    ((g007 . 0))
          rec-rel ((x . (get x SUB1))
                   (g007. (get g007 ADD1)))
          term-if (= (get EQUALS x) y)]
```

It says that the value is an (ans VAL) binding whose target can be computed by substituting for g007 the init value and then stepping through the rec-rel a-list substituting for g007 until term-if is true. This representation has the advantage that other recursive implementations (i.e. a tail-recursive one) can be mapped to an equivalent db-entry. The DSLP for diff-proc will match this db-entry instance modulo renaming and substituting (subl x) for (get x SUB1).

IV COMPARING EFFECT DESCRIPTIONS OF USER PLANS WITH LIBRARY PLANS

A variety of user programs for a given task can be recognized as having the same effect description as a pre-stored plan. The reduction to a canonical form allows the user to (1) invent arbitrary names, (2) order steps arbitrarily, (3) break up procedures arbitrarily into sub-procedures, (4) pass values through intermediate nodes of the data base, and (5) invent his own data representation.

What happens if the user's program does not match a library plan?

Our goal is to give the student as much help as possible in detecting the underlying cause of an error. Here, the effect descriptions are useful too: By using the library plan to generate the get-compositions corresponding to his data-representation, we can compare the terminal nodes of both his and the DSLP's effect description for each condition. (The assistant takes care of situations in which nested cases must be reordered for both trees to match.)

A very frequent error which can be detected by comparing effects at terminal nodes is 'name-substitution' (e.g. the student writes w and means v). The assistant compares the bindings added or deleted at each terminal branch of DSLP (goal bindings, gi) with those of the user program (actual bindings, ai). Assume it finds a mismatch of two bindings at some terminal branch:

```
goal-binding:   g = [bind (g1 g2) g3]
actual-binding: a = [bind (a1 a2) a3]
```

For example if in line 3 of DIFF-IND (Fig. 2) the code were NOTE /ANS/ VAL *T, then lines 4-6 of its effect description (Fig. 3) would be:

```
{[bind (ans VAL) (a SUBTBL)] . e}                | a
```

instead of

```
{[bind (ans VAL) (get (get a SUBTBL) b)] . e} | g
```

as suggested by 2-COL-SUB (lines 5-8 of Fig. 4).

We go up in the symbolic evaluation tree to the place where a was achieved using variable w (i.e. line 3 of DIFF-IND using variable T). If the mismatch was in pair {ai,gi}, and in the local environment some other variable was bound to gi, we can propose substituting it for the one used in the

BEST COPY AVAILABLE

code. In our example the mismatch was in {a3,g3} with a3 being T, and g3 being R, so a substitution of R for T at line 3 is suggested to the student.

If gi is not bound locally, we look at the CHECK (or procedure call) where w was bound and recursively search for the goal of binding w to gi. In case of success, we ask the student whether he wants to accept the proposed renaming.

If this fails, we try step insertion, because another typical source of errors in SOLO is students forgetting a level of indirectness (i.e. using the source of a triple, instead of the target along some link). Lines 1 and 3 in SUBTRACT (Fig. 5) manifest this error, given the implementation of DIFF-IND (Fig. 2).

A CHECK step can be inserted to correct that, if by substituting (get s K) for some subexpression s in ai (where K is some constant link), ai can be made equal to gi. In SUBTRACT the actual effect is:

```
(diff-ind B D F)                          | a
```

whereas the goal in 2-COL-SUB (lines 7-9 of Fig. 4) gives

```
(diff-ind (get B VAL) (get D VAL) F)      | g
```

and thus K = VAL. The insertion of two CHECK statements of the form CHECK B VAL ?B-VAL and modification of the calls of DIFF-IND in lines 1 and 3 (Fig. 5) is suggested.

If step insertion fails also, we look at the possibility of reversed present and absent branches (cf. [8]). This manifests itself in the effect description such that the correct data base change has occurred, but at the wrong branch of the tree. In such cases, the user is queried about whether the clauses of the CHECK statement should be reversed.

If this fails also, our last resort is to look for a branch of the tree where the conditions are correct, but there is a disagreement in the effect. The assistant chooses a counterexample that would run through this branch of the tree, and asks the student to try that example on his program. Consider line 2b in DIFF-IND (Fig. 2). The student did not consider the case where (geq a b) does not hold. The effect description for SUBTRACT (Fig. 5) has no effect where it should match lines 11-16 of 2-COL-SUB. We collect the case predicates that lead to this leaf of the symbolic evaluation tree, and see that under the condition

```
(and (present (B SUBTBL))
     (not (geq (val-of B) (val-of D))))
```

SUBTRACT would fail. Since the assistant knows that (val-of B) and (val-of D) are digits, it generates values which fulfill this condition and asks the student to try his program with those.

V CONCLUSIONS AND FURTHER WORK

Our work demonstrates how a combination of domain-specific knowledge and symbolic evaluation techniques can be applied to building a help facility for novice programmers. Writing programs in an elementary assertional data base language like SOLO is experienced by our students as a simple and natural entry point into programming. However, automatically coping with the sorts of bugs which arise through programming by side-effect requires extensions to earlier work on program understanding and debugging. The techniques we have employed give the students a fair amount of freedom in choosing their own representations, a fact which reflects one of the main goals of the course they are taking, i.e. understanding computer modelling and knowledge representation.

We are currently extending the plan library in accordance with empirical observations, in order for the debugging assistant to be capable of handling a wider range of programming tasks.

REFERENCES

[1] Adam, A. and Laurent, J.P. "LAURA, a system to debug student programs." Artificial Intelligence, 15, 1980, 75-122.

[2] Eisenstadt, M. "Artificial intelligence project." Units 3/4 of Cognitive Psychology: A third Level Course. Milton Keynes, England: Open University Press, 1978.

[3] Eisenstadt, M. and Laubsch, J. "Towards an automated debugging assistant for novice programmers." Proc. AISB-80, Amsterdam, 1980.

[4] Eisenstadt, M., Laubsch, J., and Kahney, J.H. "Creating pleasant programming environments for cognitive science students." Technical report no. 16, CAL Research Group, The Open University, Milton Keynes, England, 1981.

[5] Goldstein, I. "Understanding simple picture programs." Technical Report 294, MIT AI Lab, Cambridge, MA, 1974.

[6] Miller, M. and Goldstein, I. "Structured planning and debugging." Proc. IJCAI-77, Cambridge, MA, 1977, 773-779.

[7] Rich, C. and Shrobe, H. "Initial report on a LISP programmer's apprentice." IEEE Trans. Soft. Eng., SE-4:6, 1978, 456-466.

[8] Ruth, G. "Intelligent program analysis." Artificial Intelligence, 7, 1976, 65-68.

[9] Shrobe, H., Waters, R., and Sussman, G. "A hypothetical monologue illustrating the knowledge underlying program analysis." AI Memo 507, MIT AI Laboratory, Cambridge, MA, 1979.

[10] Waters, R. "A system for understanding mathematical FORTRAN programs." AI Memo 368, MIT AI Lab, Cambridge, MA, 1976.

BEST COPY AVAILABLE